PACE: A dynamic programming algorithm for hardware/software partitioning

Knudsen, Peter Voigt; Madsen, Jan

Published in:
Proceedings. Fourth International Workshop on Hardware/Software Co-Design (Code/CASHE `96)

Link to article, DOI:
10.1109/HCS.1996.492230

Publication date:
1996

Document Version
Publisher's PDF, also known as Version of record

PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning

Peter Voigt Knudsen and Jan Madsen
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Abstract.

This paper presents the PACE partitioning algorithm which is used in the LYCOS co-synthesis system for partitioning control/dataflow graphs into hardware- and software parts. The algorithm is a dynamic programming algorithm which solves both the problem of minimizing system execution time with a hardware area constraint and the problem of minimizing hardware area with a system execution time constraint. The target architecture consists of a single microprocessor and a single hardware chip (ASIC, FPGA, etc.) which are connected by a communication channel. The algorithm incorporates a realistic communication model and thus attempts to minimize communication overhead. The time-complexity of the algorithm is $O(n^2 \cdot \mathcal{A})$ and the space-complexity is $O(n \cdot \mathcal{A})$ where $\mathcal{A}$ is the total area of the hardware chip and n the number of code fragments which may be placed in either hardware or software.

1 Introduction

The hardware/software partitioning of a system specification onto a target architecture consisting of a single CPU and a single ASIC has been investigated by a number of research groups [2, 5, 1, 7, 8, 11]. This target architecture is relevant in many areas where the performance requirements cannot be met by general-purpose microprocessors, and where a complete ASIC solution is too costly. Such areas may be found in DSP design, construction of embedded systems, software execution acceleration and hardware emulation and prototyping [10].

One of the major differences among partitioning approaches is in the way communication between hardware and software is taken into account during partitioning. Henkel, Ernst et al. [2, 1, 6] present a simulated annealing algorithm which moves chunks of software code (in the following called blocks) to hardware until timing constraints are met. The algorithm accounts for communication and only variables which need to be transferred are actually taken into account, i.e., the possibility of local store is exploited. Gupta and De Micheli [5] present a partitioning approach which starts from an all-hardware solution. Their algorithm takes communication into account and is able to reduce communication when neighboring vertices are placed together in either software or hardware. The system model presented by Jantsch et al. [7] ignores communication. They present a dynamic programming algorithm based on the Knapsack algorithm which solves the partitioning problem for the case where some blocks include other blocks and are therefore mutually exclusive. The algorithm has exponential memory requirements which makes it impractical to use for large applications. To solve this problem they propose a pre-selection scheme which only selects blocks which induce a speedup greater than 10%. However, this pre-selection scheme may fail to produce good results as communication overhead is ignored. Kalavade and Lee [8] present a partitioning algorithm which does take communication into account by attributing a fixed communication time to each pair of blocks. This approach may overestimate the communication overhead as more variables than actually needed are transferred.

In this paper we present a dynamic programming algorithm called PACE [9] which solves the hardware/software partitioning problem taking communication overhead into account.

2 System Model

This section presents the system model used by the partitioning algorithm, and describes how it is obtained from the functional specification.

2.1 The CDFG Format

The functional specification, which is currently described in VHDL or C, is internally represented as a control/data flow graph (CDFG) which can be defined as follows:

Definition 1 *A CDFG is a set of nodes and directed edges (N, E) where an edge $e_{i,j} = (n_i, n_j)$ from $n_i \in N$ to $n_j \in N$, $i \neq j$, indicates that n_j depends on n_i because of data dependencies and/or control dependencies.*

Definition 2 *A node $n_i \in N$ is recursively defined as*

$$n_i = \text{DFG} \mid \text{Cond} \mid \text{Loop} \mid \text{FU} \mid \text{Wait}$$

0-8186-7243-9/96 $05.00 © 1996 IEEE

$$
\begin{aligned}
\text{Cond} &= (\text{Branch1}, \text{Branch2}) \\
\text{Loop} &= (\text{Test}, \text{Body}) \\
\text{Branch1} &= \text{CDFG} \\
\text{Branch2} &= \text{CDFG} \\
\text{Test} &= \text{CDFG} \\
\text{Body} &= \text{CDFG} \\
\text{FU} &= \text{CDFG}
\end{aligned}
$$

where a DFG *is a pure dataflow graph without control structures,* FU *represents a function call,* Wait *is used for synchronization with the environment,* Branch1 *and* Branch2 *are the CDFGs to be executed in the "true" and "false" branch case of a conditional* Cond *respectively and* Test *and* Body *are the test and body CDFGs of a* Loop.

2.2 Derivation of Basic Scheduling Blocks

In order to be able to partition the CDFG, it must first be divided into fragments, in the following called Basic Scheduling Blocks or BSBs. For each node n_i in the CDFG, a BSB is created as shown in figure 1. Each

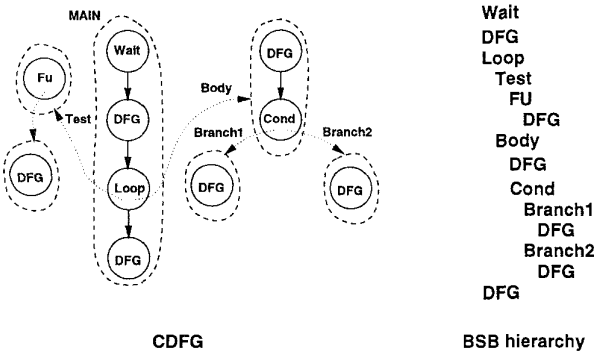

Figure 1: *The BSB hierarchy and its correspondence with the CDFG.*

BSB can have child BSBs which are shown indented under the BSB. In this way a *BSB hierarchy* which reflects the hierarchy of the CDFG is obtained. A BSB stores information which is used by the partitioning algorithm to determine whether it should be placed in hardware or software:

Definition 3 *A BSB, B_i, is defined as a six-tuple*

$$B_i = < a_{s,i}, t_{s,i}, a_{h,i}, t_{h,i}, r_i, w_i >$$

where $a_{s,i}$ and $t_{s,i}$ are the area and execution time of B_i when placed in software, $a_{h,i}$ and $t_{h,i}$ are the area and execution time of B_i when placed in hardware and r_i and w_i contain the read-set and write-set variables of B_i.

In order to be able to control the number and sizes of the BSBs which are considered by the partitioning algorithm, parent BSBs can be collapsed as to appear as single BSBs instead of the child BSBs they are composed of. This is illustrated in figure 2. The partition-

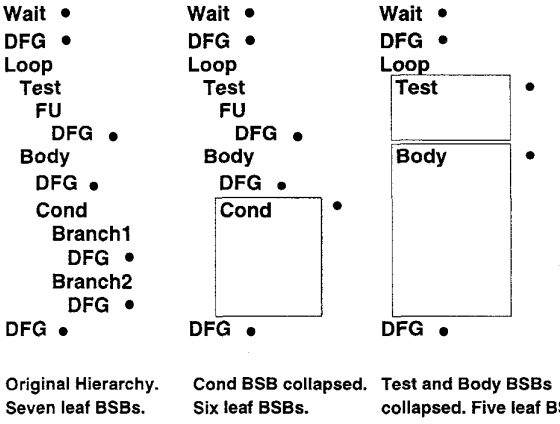

Figure 2: *Adjusting BSB granularity by hierarchical collapsing.*

ing algorithm only considers *leaf BSBs* which are BSBs which have no children. The leaf BSBs are marked with a dot in the figure. When BSBs are collapsed, the number of leaf BSBs decreases. As the execution time of the PACE algorithm depends quadratically on the number of BSBs, it is relevant to be able to control the number of BSBs in this way.

As all leaf BSBs together make up the total system functionality, we can now define the system specification in terms of leaf BSBs:

Definition 4 *A system specification S is described as an ordered list of n leaf BSBs, i.e. $\{B_1, B_2, \ldots, B_n\}$ where B_i denotes BSB number i.*

In order to estimate performance, it is necessary to know how many times each BSB is executed for typical input data. This information is obtained from profiling. It is convenient to define two global functions which return profiling information for individual BSBs and individual variables:

Definition 5 *The function* pc : $B_i \in S \rightarrow Nat$ *returns the number of times B_i has been executed in a profiling run*[1].

Definition 6 *Let V denote the set of all variables from the read-sets and write-sets of the BSBs in S. Then, for a given variable v from the read-set or write-set of B_i, the function* ac : $v \in V \rightarrow Nat$ *returns the number of times the variable is accessed by B_i*:[2]

$$(v \in r_i) \vee (v \in w_i) \Rightarrow \text{ac}(v) = \text{pc}(B_i)$$

[1] "pc" is short for "profiling count".
[2] "ac" is short for "access count".

3 Problem formulation

The partitioning model which the PACE algorithm uses is illustrated in figure 3. In this model hardware

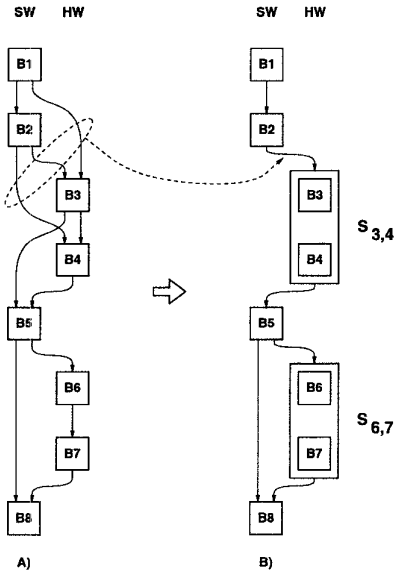

Figure 3: *Partitioning model used by PACE: a) Example of actual data-dependencies between hardware- and software BSBs, b) How data-dependencies between adjacent hardware BSBs and software BSBs are interpreted in the model.*

BSBs and software BSBs cannot execute in parallel. Furthermore, adjacent hardware BSBs are assumed to be able to communicate the read/write variables they have in common directly between them without involving the software side. As illustrated in the figure, a given hardware/software partition can be thought of as composed of sequences of adjacent BSBs which only communicate their *effective* read- and write-sets from/to the software side. The following definitions formalize these assumptions.

Definition 7 *$S_{i,j}$, $j \geq i$, denotes the sequence of BSBs $\{B_i, B_{i+1}, \ldots, B_j\}$.*

Definition 8 *The effective read-set and the effective write-set of a sequence $S_{i,j}$ are denoted $r_{i,j}$ and $w_{i,j}$ respectively and are defined as*

$$\begin{aligned} r_{i,j} &= (r_i \cup r_{i+1} \cup \cdots \cup r_j) \setminus (w_i \cup w_{i+1} \cup \cdots \cup w_j) \\ w_{i,j} &= (w_i \cup w_{i+1} \cup \cdots \cup w_j) \setminus (r_i \cup r_{i+1} \cup \cdots \cup r_j) \end{aligned}$$

Using these definitions and the BSB definitions given in section 2.2 we can now compute the speedup induced by moving a sequence of BSBs from hardware to software:

Definition 9 *The total (possibly negative) speedup induced by moving a BSB sequence $S_{i,j}$ to hardware is denoted $s_{i,j}$ and is computed as*

$$\begin{aligned} s_{i,j} = & \sum_{k=i}^{j} \mathrm{pc}(B_k)(t_{s,k} - t_{h,k}) - \\ & (\sum_{v \in r_{i,j}} \mathrm{ac}(v) t_{s \to h} + \sum_{v \in w_{i,j}} \mathrm{ac}(v) t_{h \to s}) \end{aligned}$$

where $t_{s \to h}$ and $t_{h \to s}$ denote the software-to-hardware and hardware-to-software communication times for a single variable, respectively.

Definition 10 *The area penalty $a_{i,j}$ of moving $S_{i,j}$ to hardware is computed as the sum of the individual BSB areas:*

$$a_{i,j} = \sum_{k=i}^{j} a_k$$

In section 4.2 we discuss how the effect of hardware sharing is taken into account. Note that in calculating the speedup and area of a sequence it is not considered that hardware synthesis may synthesize the sequence as a whole which would probably reduce both sequence area and execution time as compared to just summing the individual area- and execution time components as described above. Incorporating such sequence optimizations in the estimations will be fairly straightforward but has not been carried out yet. Note, however, that the improvement in speedup induced by all BSBs within the sequence being able to communicate directly with each other *is* taken into account.

The partitioning problem can now be formulated as that of finding the combination of non-overlapping hardware sequences which yields the best speedup while having a total area penalty less than or equal to the available hardware area $\mathcal{A}$.

4 Software, Hardware and Communication Estimation

This section describes how hardware area- and execution time, software execution time, and communication time are estimated.

4.1 Software Estimation

Software execution time for a pure DFG (i.e., no controlflow) is estimated by performing a topological sort (linearization) of the nodes in the DFG. The nodes are then translated into a generic instruction set with the addressing modes of the instructions determined by data-dependencies and a greedy register allocation scheme. The execution times of the generic instructions

are then determined from a technology file corresponding to the target microprocessor. This is similar to the approach described in [3], where good estimation results are reported, and the same technology files for the 8086, 80286, 68000 and 68020 microprocessors are used. The execution time of the DFG is obtained by summing the execution times of the generic instructions and multiplying the sum with the profiling count for the DFG. Execution times for higher level constructs such as loop BSBs and branch BSBs are obtained on basis of the execution times of their child BSBs.

4.2 Hardware area estimation

A common way of estimating the hardware area of a BSB is to estimate how much area a full hardware implementation of the BSB will occupy. This includes hardware to execute the calculations of the BSB and hardware to control the sequencing of these calculations. If the total chip area is divided into a *datapath area* and a *controller area*, each BSB moved to hardware may be viewed as occupying a part of the datapath and a part of the controller. Figure 4a shows this model when one BSB has been moved to hardware.

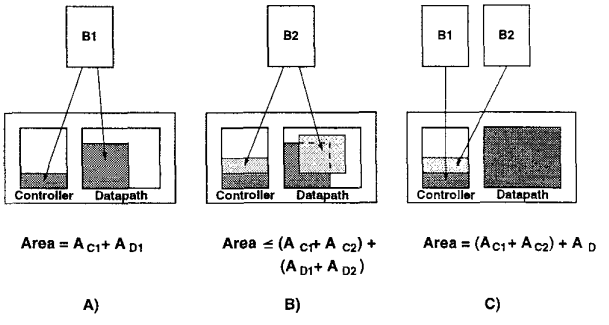

Figure 4: *BSB area estimation which accounts for hardware sharing: a) Controller- and datapath area for a single BSB, b) When sharing hardware, the total area for multiple BSBs are less than the summation of the individual areas, c) Variable controller area and fixed datapath area for multiple BSBs with hardware sharing.*

When several BSBs are moved to hardware they may share hardware modules as they execute in mutual exclusion. Hence, an approach which estimates area as the summation of datapath and control areas for all hardware BSBs will probably overestimate the total area. This problem is depicted in figure 4b where the area of the datapath is *not* equal to the sum of the individual BSB datapaths.

In our approach the datapath area is the area of a set of preallocated hardware modules in the datapath as illustrated in figure 4c. The BSBs share these modules and the controller area is the area left for the BSB controllers. The hardware area of a BSB is then estimated as the hardware area of the corresponding controller, and will depend on the number of timesteps required for executing the BSB. The hardware area of a BSB therefore depends on its execution time.

4.3 Hardware execution time estimation

The hardware execution time for a DFG is determined by dynamic list based scheduling [4] which attempts to utilize the hardware modules in the given allocation in order to maximize parallelism and thereby minimize execution time. The execution time obtained in this way is multiplied with the profiling count for the DFG. The execution time for higher level constructs is obtained as in the software case.

4.4 Communication estimation

Communication is currently assumed to be memory mapped I/O. The transfer of k variables from software to hardware is assumed to require k generic `MOV` microprocessor instructions and k `Import` operations as defined in the hardware library. Communication from hardware to software is estimated in the same way, just using the hardware `Export` operation instead.

5 The PACE algorithm

The idea behind the PACE algorithm is best illustrated by an example. Figure 5 shows four BSBs which must be partitioned as to reach the largest speedup on the available area $\mathcal{A}$=3. The speedup and area penalty for a single BSB which is moved to hardware is shown below each BSB. The numbers between two BSBs denote the extra speedup which is incurred because of the BSBs being able to communicate directly with each other when they are both placed in hardware.

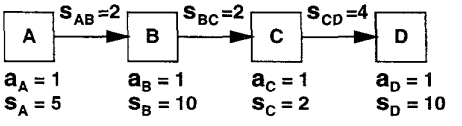

Figure 5: *Example of partitioning problem with communication cost considerations.*

Obviously B and D should be placed in hardware as they have large inherent speedups. This leaves room for one more BSB. Should it be A or C? The answer to this is not obvious as A induces a large inherent speedup but a small communication speedup when placed together with B in hardware, whereas C induces a smaller inherent speedup but on the other hand induces a large communication speedup when placed together with B and D in hardware. The following paragraphs explain how the PACE algorithm solves this problem.

The algorithm utilizes the previously mentioned fact, that any possible partition can be thought of as

composed of *sequences* of BSBs. If A, C and D are chosen for hardware, it corresponds to choosing the sequences $S_{A,A}$ and $S_{C,D}$. The speedup of sequence $S_{C,D}$ is larger than the sum of speedups of its components C and D due to the extra communication speedup induced by both blocks being chosen for hardware. So a natural approach will be to calculate the areas and speedups of *all sequences* of BSBs, and chose the combination of sequences that induces the largest speedup. The areas and speedups of all sequences are calculated and shown in table 1. The ordering and grouping of BSBs is explained below.

Sequence	Elements	Area	Speedup
Group A: All sequences ending with A			
$S_{A,A}$	A	1	5
Group B: All sequences ending with B			
$S_{A,B}$	AB	2	17
$S_{B,B}$	B	1	10
Group C: All sequences ending with C			
$S_{A,C}$	ABC	3	21
$S_{B,C}$	BC	2	14
$S_{C,C}$	C	1	2
Group D: All sequences ending with D			
$S_{A,D}$	ABCD	4	35
$S_{B,D}$	BCD	3	28
$S_{C,D}$	CD	2	16
$S_{D,D}$	D	1	10

Table 1: *Grouping of sequences.*

The problem is to find the combination of non-overlapping sequences which fits the available area $\mathcal{A}$ and whose speedup sum is as large as possible. This problem cannot be solved with an ordinary Knapsack Stuffing algorithm as some of the sequences are mutually exclusive (because they contain identical BSBs) and therefore cannot be moved to hardware at the same time. But if the sequences are ordered and grouped as shown in the table, a dynamic programming algorithm *can* be constructed which does not attempt to chose mutually exclusive sequences for hardware at the same time.

The algorithm works as follows. Assume first that for each group up to and including group C the best (maximum speedup) combination of sequences has been found and stored for each (integer) area a from zero up to the available area $\mathcal{A}$. Assume then that for instance sequence $S_{C,D}$ with area $a_{C,D}$ is selected for hardware at the available area a. How is the optimal combination of sequences on the remaining area $a - a_{C,D}$ then found? As C and D have been chosen for hardware, only A and B remain. So the best solution on the remaining area must be found in group B which contains the best combination of sequences for all BSBs from the set {A,B}. Similarly, if the "sequence" $S_{D,D}$ is chosen for hardware, the best combination on the remaining area is found in group C. The optimal combination is always found in the group whose letter in the alphabet comes immediately before the letter of the first index in the chosen sequence. The important thing to note is that when a sequence from

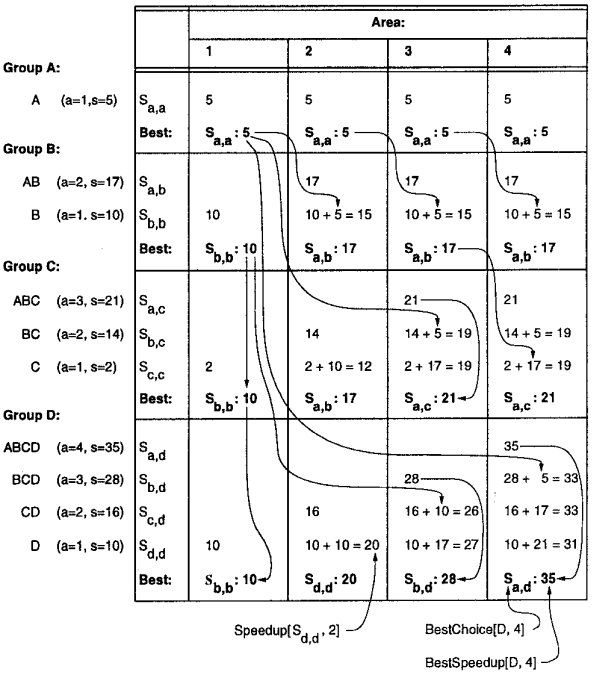

Figure 6: *The PACE algorithm employed for a simple example.*

group X has been chosen, the optimal combination of sequences on the remaining area can be found in one of the groups A to `pred`(X), and, when sequences are selected as above, no mutually exclusive BSBs are selected simultaneously. In this way the best solutions for a given group can always be determined on basis of the best solutions found for the previous groups.

Figure 6 shows how the best combination of sequences can be found using three matrices; `Speedup[1..`n_S`, 0..`$\mathcal{A}$`]`, `BestSpeedup[1..`n`,1..`$\mathcal{A}$`]` and `BestChoice[1..`n`, 0..`$\mathcal{A}$`]`.

n_S is the number of sequences, n is the number of BSBs and $\mathcal{A}$ is the available area. Zero entries are not shown. Arrows indicate where values are copied from, but arrows are not shown for all entries in order to make the figure more readable.

The `Speedup` matrix contains for each sequence and each available area the best speedup that can be achieved if that sequence is first moved to hardware and then sequences from the *previous* groups are moved to hardware. In the figure, `Speedup[`$S_{B,C}$`, 3]` is 19 and is found as the inherent speedup of $S_{B,C}$ which is 14 plus the best obtainable speedup 5 on area $3 - a_{B,C} = 3 - 2 = 1$ in group A (as B and C have been chosen). The `BestSpeedup` matrix contains for each group (which there are n of) and each area the best speedup that can be achieved by first selecting a sequence from that group or one of the previous groups. It can be calculated as

`BestSpeedup[`g`, `a`] =`

$$\max\left(\max_{S \in g}(\texttt{Speedup}[S,a]), \texttt{BestSpeedup}[\texttt{pred}(g),a]\right)$$

The `BestChoice[`g`,`a`]` matrix identifies the choice of sequence that gave this maximum value. The last two matrices are interleaved and typeset with bold letters in the figure.

In the example, `BestChoice[C, 3]` is 21 as this is the maximum speedup that can be found in group C with available area 3 and it is larger than the largest speedup that could be found in the previous groups, namely 17. The corresponding choice of sequence is $S_{A,C}$. In contrast, `BestSpeedup[C,1]` and `BestChoice[D,1]` are copied from the corresponding entries of the previous group. For group C this is because all `Speedup` entries in that group for area 1 are smaller than the best speedup 10 achieved with only sequences from groups up to and including B. For group D, `Speedup[`$S_{D,D}$`, 1]` is also 10, so the choice of best sequence for this group is arbitrary.

The solution to the posed problem is found in the `BestChoice[D, 3]` and `BestSpeedup[D,3]` entries. The best initial choice is sequence $S_{B,D}$ with the corresponding total speedup of 28. As the area of this sequence is 3, no other sequences were taken, and need thus not be found by backtracking. This shows that it was best to chose C for hardware instead of A. The area 4 was included in the figure to show that the algorithm correctly chooses all four BSBs for hardware when there is room for them. This can be seen from the [D,4] entries.

Once the `BestSpeedup` and `BestChoice` lines have been calculated for each group, the `Speedup` values are no longer needed. Actually, the `Speedup` matrix is not needed at all, as it can be replaced by the `BestSpeedup` matrix whose maximum values can be calculated "on the run". This is because we are only interested in maximum values and corresponding choice of sequences for each group. This means that instead of the memory requirements being proportional to the number of sequences n_S, they are now proportional to n, as only the `BestSpeedup` and `BestChoice` matrices are needed. The PACE algorithm is shown as algorithm 1.

After the algorithm has been run, the best speedup that can be obtained is found in the entry `BestSpeedup[NumBSBs, AvailableArea]`. But as for the simple Knapsack algorithm, reconstruction of the chosen sequences and thereby of the chosen BSBs is necessary.

5.1 Algorithm Analysis

Direct inspection of the PACE algorithm shows that the time complexity is $O(n^2 \cdot \mathcal{A})$ and the space complexity is $O(n \cdot \mathcal{A})$ (the PACE-reconstruct algorithm obviously has smaller time and area complexity and can hence be disregarded). Note that areas must be expressed as integral values. $\mathcal{A}$ can be reduced (at the expense of partitioning quality) by using a larger "area granularity", for example by expressing BSBs sizes in

```
PACE (n, A) ≡
{
   for_all groups g = 1 to n do
      for_all areas a = 0 to A do {
         BC[g,a] ← {};
         BS[g,a] ← 0;
      }
   }
   for_all groups g = 1 to n do {
      HighBSB ← g;
      for LowBSB = 1 to HighBSB do {
         SeqArea ← area(S_LowBSB,HighBSB);
         SeqSpeedup ← speedup(S_LowBSB,HighBSB);
         for_all areas a = SeqArea to A do {
            if (LowBSB = 1) then {
               if SeqSpeedup > BS[g, a] then {
                  BS[g,a] ← SeqSpeedup;
                  BC[g,a] ← S_LowBSB,HighBSB;
               }
            } else {
               if SeqSpeedup + BS[LowBSB-1, a-SeqArea] >
                  BS[g, a] then {
                  BS[g,a] ← SeqSpeedup +
                     BS[LowBSB-1, a-SeqArea];
                  BC[g,a] ← S_LowBSB,HighBSB;
               }
            }
         }
      }
      if (HighBSB > 1)
         for_all areas a = 0 to A do
            if BS[g-1, a] > BS[g, a] then {
               BS[g, a] ← BS[g-1, a];
               BC[g, a] ← BC[g-1, a];
            }
   }
   return BC[];
   return BS[];
}

PACE-reconstruct (n, A, BS[], BC[]) ≡
{
   HwBSBList ← {};
   AStart ← 0;
   Found ← false;
   while (AStart <= A) and not Found do
      if BS[n, AStart] = BS[n, A] then
         Found ← true
      else
         AStart ← AStart + 1;
   a ← Astart;
   g ← n;
   repeat {
      Seq ← BC[g, a];
      if Seq <> {} then {
         LowBSB ← first index of Seq;
         HighBSB ← second index of Seq;
         for BSB = LowBSB to HighBSB do
            add BSB to HwBSBList;
         a ← A − area(Seq);
         g ← LowBSB − 1;
      }
   } until (a < 0) or (Seq = {}) or (g = 0);
   return HwBSBList;
}
```

Algorithm 1: *PACE - A Partitioning Algorithm with Communication Emphasis.*

terms of RTL modules. The n^2 term can be reduced by enlarging BSB granularity by hierarchical collapsing or by only considering BSB sequences which induce a speedup greater than zero (or greater than some given percentage). Also, there is no need to pre-calculate areas and speedups for sequences whose total area will be greater than $\mathcal{A}$. Another optimization could be to only consider sequences of length $\leq m$ where m is an arbitrary value selected prior to execution of the algorithm.

As for the simple Knapsack problem, the dual problem of minimizing area with a fixed time-constraint can be solved by swapping the area- and speedup entries calculated for each sequence. Another approach could be to scan the bottom line of the `BestSpeedup` matrix from the left (see figure 6) until an entry is found which violates the time-constraint, as the PACE algorithm in effect calculates the best speedup for *all* areas.

Note that the areas and speedups of all sequences must be pre-calculated before the algorithm can execute. This operation basically has time complexity $O(n^2)$ but this can also be reduced.

6 Experiments

A series of experiments which demonstrate the capabilities of the PACE algorithm have been carried out. The experiments were carried out in LYCOS, the LYngby CO-Synthesis system, an experimental co-synthesis system currently being developed at our institute.

The sample application used for the experiments is a VHDL behavioral description, taken from an image processing application in optical-flow. The application calculates eigenvectors which are used to obtain local orientation estimates for the cloud movements in a sequence of Meteosat thermal images. The application consisting of 448 lines of behavioral VHDL. The corresponding CDFG contains 1511 nodes and 1520 edges. BSB software execution-time is estimated for a 8086 processor and a hardware library for an Actel ACT 3 FPGA is used to estimate hardware datapath and controller area. Partitioning is performed for a sequence of total hardware areas ranging from 1000 to 2000 in steps of 20, where an area unit equals the area of a logic/sequential module in the FPGA. Table 2 summarizes the characteristics of the most important modules.

Hardware modules used for the experiments			
Module	*Operations*	*Area*	*Cycles*
add-sub-comb	add, subtract, less, equal	97	1
mul-comb	multiply	339	4
mul-ser	multiply	103	16
div-comb	divide	339	4
div-ser	divide	103	16

Table 2: *Area and execution-time estimates for hardware modules and operations.*

Figure 7 shows the results of partitioning the sample application using three different partitioning models;

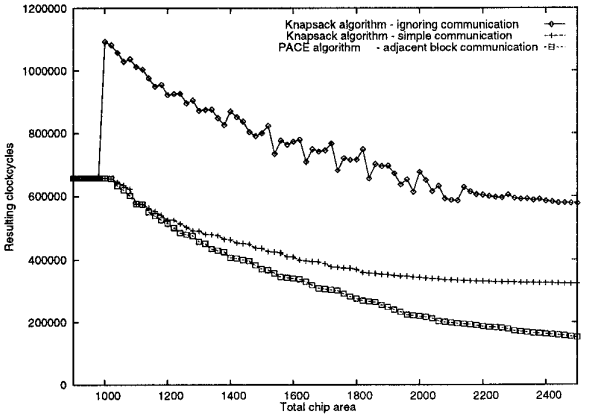

Figure 7: *The PACE algorithm compared with Knapsack algorithms which ignore communication or do not account for adjacent hardware block communication optimization (allocation A).*

ignoring communication, simple communication where the read- and write-sets of a BSB are *always* transferred regardless of other BSBs placed in hardware, and adjacent block communication which is the one used by the PACE algorithm. All three approaches are assumed to be implemented with local hardware store, and are thus evaluated in the model domain according to the adjacent block communication model.

For chip areas less that the allocated area for the datapath (in the figure 760), no speedup is obtained as no control area is available. As soon as control area is available the approach which ignores communication starts to move BSBs to hardware. For the approaches taking communication into account, moving BSBs to hardware is not beneficial before the total area reaches around 1040. It can be seen that as the chip area increases, more and more BSBs are moved to hardware. From the figure it is clear that the approach using the simple communication model does not move as many BSBs to hardware as the approach which takes adjacent block communication into account. This is mainly due to the fact that many of the BSBs have a communication overhead which is larger than the speedup they induce. In any case the best results are obtained by partitioning according to the adjacent block communication model.

Datapath allocations			
	Allocation		
Module	*A*	*B*	*C*
add-sub-comb	2	1	1
mul-comb	1	0	0
mul-ser	0	8	1
div-comb	1	0	0
div-ser	0	1	1
Area	760	1148	427

Table 3: *Modules and corresponding area for each of the three allocations.*

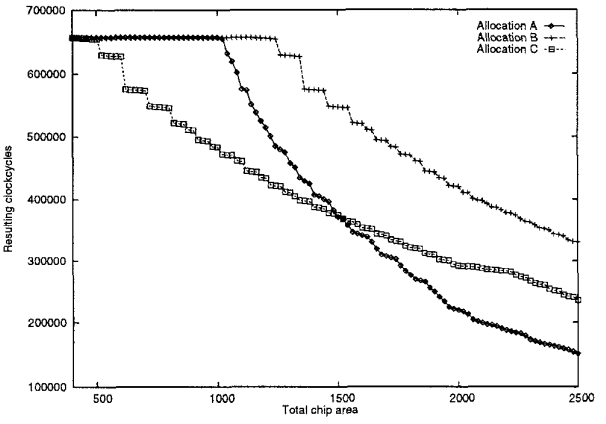

Figure 8: *The PACE algorithm employed for different allocations.*

Figure 8 shows the results of partitioning with the PACE algorithm for three different allocations; A, B and C, all listed in table 3.

Widely different results are obtained for given available areas, and a specific allocation which is optimal for all areas cannot be found. Allocation C has the smallest datapath area which means that for relatively small areas, the partitioning algorithm is able to move BSBs to hardware and, thus, obtain the best partition. Around the area 1500 this is changed, now allocation A becomes more attractive due to the fact that the larger datapath allocation can benefit from the inherent parallelism of the sample application, i.e., larger speedups may be achieved for the individual BSBs. The figure also illustrates the problem of allocating too much datapath area, as allocation B which has the largest datapath area never manages to give the best partitioning even for large chip areas.

7 Conclusion

This paper has presented the PACE hardware/software partitioning algorithm which, within the presented partitioning model, gives an exact solution for the problem of minimizing total execution time with a hardware area constraint as well as for the problem of minimizing hardware area with a global execution time constraint. The partitioning model recognizes 1) that sequences of hardware BSBs only need to communicate their effective read- and write sets from/to software and 2) that both area and execution times can be calculated on a BSB sequence basis and therefore may be smaller than the sum of the individual area- and execution times of the BSBs in the sequence. The algorithm has quadratic time complexity, but as shown, this may be reduced.

Experiments have been carried out that show that the algorithm is superior to algorithms which ignore communication or do not recognize adjacent block communication optimization . Also, it has been demonstrated how the algorithm can be used with a BSB hardware area model which assumes hardware sharing among BSBs.

Acknowledgements

This work is supported by the Danish Technical Research Council under the "Codesign" program.

References

[1] R. Ernst, J. Henkel, and T. Benner. Hardware/software co-synthesis of microcontrollers. *Design and Test of Computers*, pages 64–75, December 1992.

[2] Rolf Ernst, Wei Ye, Thomas Benner, and Jörg Henkel. Fast timing analysis for hardware/software co-design. In *ICCD '93*, 1993.

[3] Jie Gong, Daniel D. Gajski, and Sanjiv Narayan. Software estimation from executable specifications. Technical Report ICS-93-5, Dept. of Information and Computer Science, University of California, Irvine, Irvine, CA 92717-3425, March 8 1993.

[4] Jesper Grode. Scheduling of control flow dominated data-flow graphs. Master's thesis, Technical University of Denmark, 1995.

[5] Rajesh K. Gupta and Giovanni De Micheli. System synthesis via hardware-software co-design. Technical Report CSL-TR-92-548, Computer Systems Laboratory, Stanford University, October 1992.

[6] D. Herrmann, J. Henkel, and R. Ernst. An approach to the adaptation of estimated cost parameters in the cosyma system. In *CODES '94*, 1994.

[7] Axel Jantsch, Peeter Ellervee, Johnny Öberg, Ahmed Hermani, and Hannu Tenhunen. Hardware/software partitioning and minimizing memory interface traffic. In *EURO-DAC '94*, 1994.

[8] Asawaree Kalavade and Edward A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Third International Workshop on Hardware/Software Codesign*, pages 42–48, September 1994.

[9] Peter V. Knudsen. Fine-grain partitioning in codesign. Master's thesis, Technical University of Denmark, 1995.

[10] Giovanni De Micheli. Computer-aided hardware-software codesign. *IEEE Micro*, 14(4):10–16, August 1994.

[11] Frank Vahid, Jie Gong, and Daniel D. Gajski. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In *EURO-DAC '94*, pages 214–219, 1994.